Государственное образовательное учреждение
высшего профессионального образования
«Московский государственный технический университет
имени Н.Э. Баумана»

Отчет
По лабораторной работе №4
По курсу «Конструирование компиляторов»
На тему
«Синтаксический анализатор операторного предшествования»

Студент: Мурашов И.Д.
Группа: ИУ7-23М
Вариант: 1
Преподаватель: Ступников А.А.

Москва, 2020

Оглавление

1 Цель и задачи работы

Цель работы: приобретение практических навыков реализации таблично управляемых синтаксических анализаторов на примере анализатора операторного предшествования.
Задачи работы

1. Ознакомиться с основными понятиями и определениями, лежащими в основе синтаксического анализа операторного предшествования.

2. Изучить алгоритм синтаксического анализа операторного предшествования.

3. Разработать, тестировать и отладить программу синтаксического анализа в соответствии с предложенным вариантом грамматики.

4. Включить в программу синтаксического анализа семантические действия для реализации синтаксически управляемого перевода инфиксного выражения в обратную польскую нотацию.

Исходная грамматика:
<выражение> -> <простое выражение> | <простое выражение> <операция отношения> <простое выражение>
<простое выражение> -> <терм> | <знак> <терм> | <простое выражение> <операция типа сложения> <терм>
<терм> -> <фактор> | <терм> <операция типа умножения> <фактор>
<фактор> -> <идентификатор> | <константа> | (< простое выражение >) | not <фактор>
<операция отношения> -> = | <> | < | <= | > | >=
<знак> -> + | -
<операция типа сложения> -> + | - | or
<операция типа умножения> -> * | / | div | mod | and

Замечания.

1. Нетерминалы <идентификатор> и <константа> - это лексические единицы (лексемы), которые оставлены неопределенными, а при выполнении лабораторной работы можно либо рассматривать их как терминальные символы, либо определить их по своему усмотрению и добавить эти определения.

2. Терминалы not, or, div, mod, and - ключевые слова (зарезервированные).

3. Терминалы ( ) - это разделители и символы пунктуации.

4. Терминалы = <> < <= > >= + - * / - это знаки операций.

5. Нетерминал <выражение> - это начальный символ грамматики.

2 Текст программы

```
        using Lab4_Precedence.Elements;
        using System;
        using System.Collections.Generic;
        using System.Linq;

        namespace Lab4_Precedence
        {
        public class PrecedenceParser
        {
        private readonly Symbol GrammarStart = new Symbol("E");

        private readonly Operator[] Ops = new Operator[] { new Operator("not"
        new Operator("div", 3, Operator.Assoc.Left), new Operator("mod", 3, O
        new Operator("+", 2, Operator.Assoc.Left), new Operator("-", 2, Opera
new Operator("<", 1, Operator.Assoc.Left),
        new Operator("<=", 1, Operator.Assoc.Left), new Operator("=", 1, Oper
        new Operator(">", 1, Operator.Assoc.Left), new Operator(">=", 1, Oper

        private readonly Symbol[] Vars = new Symbol[] { new Symbol("a"),
new Symbol("c") };
        private readonly Symbol[] Parens = new Symbol[] { new Symbol("("), ne
        private readonly Symbol Marker = new Symbol("$");

        private List<Symbol> symbols;
        private List<List<char>> precedenceTable;

        public PrecedenceParser()
        {
        symbols = Vars.Concat(Parens).Concat(Ops).ToList();
        symbols.Add(Marker);

        precedenceTable = new List<List<char>>(symbols.Count);
        for (int i = 0; i < symbols.Count; i++)
        {
        precedenceTable.Add(new List<char>(symbols.Count));
        for (int j = 0; j < symbols.Count; j++)
        {
        if (IsOper(symbols[i]))
        {
        if (IsOper(symbols[j]))
        {
        if (((Operator)symbols[i]).Prior > ((Operator)symbols[j]).Prior)
        precedenceTable[i].Add('>');
        else if (((Operator)symbols[i]).Prior < ((Operator)symbols[j]).Prior)
        precedenceTable[i].Add('<');
        else if (((Operator)symbols[i]).Ass == Operator.Assoc.Left)
        precedenceTable[i].Add('>');
        else if (((Operator)symbols[i]).Ass == Operator.Assoc.Right)
```

```
precedenceTable[i].Add('<');
else
precedenceTable[i].Add(' ');
}
else if (IsVar(symbols[j].Symb) || symbols[j].Symb == Parens[0].Symb)
{
precedenceTable[i].Add('<');
}
else if (symbols[j].Symb == Parens[1].Symb || symbols[j].Symb == Marl
{
precedenceTable[i].Add('>');
}
else
{
precedenceTable[i].Add(' ');
}
}
else if (IsVar(symbols[i].Symb))
{
if (IsOper(symbols[j]) || symbols[j].Symb == Parens[1].Symb || symbol
precedenceTable[i].Add('>');
else
precedenceTable[i].Add(' ');
}
else if (symbols[i].Symb == Parens[0].Symb)
{
if (IsOper(symbols[j]) || symbols[j].Symb == Parens[0].Symb || IsVar(
precedenceTable[i].Add('<');
else if (symbols[j].Symb == Parens[1].Symb)
precedenceTable[i].Add('=');
else
precedenceTable[i].Add(' ');
}
else if (symbols[i].Symb == Parens[1].Symb)
{
if (IsOper(symbols[j]) || symbols[j].Symb == Parens[1].Symb || symbol
precedenceTable[i].Add('>');
else
precedenceTable[i].Add(' ');
}
else if (symbols[i].Symb == Marker.Symb)
{
if (IsOper(symbols[j]) || symbols[j].Symb == Parens[0].Symb || IsVar(
precedenceTable[i].Add('<');
else
precedenceTable[i].Add(' ');
}
}
}
```

```
TableOutput();
}

public string ParseToPostfix(string[] input)
{
string postfix = "";
Stack<string> vars = new Stack<string>();

string[] tokens = ReplaceUnar(input);

tokens = tokens.Append(Marker.Symb).ToArray();
Stack<Symbol> stack = new Stack<Symbol>();
stack.Push(Marker);

int currToken = 0;

while (tokens[currToken] != Marker.Symb || !IsStackDone(stack))
{
//if (IsVar(tokens[currToken]))
//     vars.Push(tokens[currToken]);

int headStackToken = symbols.FindIndex(s=> s.Symb == NotStartSymbolHe
int incomeToken = symbols.FindIndex(s => s.Symb == tokens[currToken])

char relation;
try
{
relation = precedenceTable[headStackToken][incomeToken];
}
catch (ArgumentOutOfRangeException)
{
Console.WriteLine("Wrong token at {0} pos!", currToken + 1);
return null;
}

if (relation == '<' || relation == '=')
{
stack.Push(symbols.Find(s => s.Symb == tokens[currToken]));

currToken++;
}

else if (relation == '>')
{
bool flag = true;
for (int i = 1; i < stack.Count && flag; i++)
{
List<Symbol> stackCut = stack.Take(i).ToList();
if (stackCut.Count == 1)
{
```

```
if (IsVar(stackCut[0].Symb))
{
var a = stack.Pop();

stack.Push(new Symbol(GrammarStart.Symb, a.Symb));

flag = false;
}
}
else if (stackCut.Count == 2)
{
if (IsUnarOper(stackCut[1]) && stackCut[0].Symb == GrammarStart.Symb)
{
string postfixAdd = "";
if (stackCut[0].Value != "")
{
postfixAdd = stackCut[0].Value + stackCut[1].Symb;
}
//else
//postfixAdd = stackCut[1].Symb;

postfix += postfixAdd;

stack.Pop(); stack.Pop();
stack.Push(GrammarStart);

flag = false;
}

if (IsNot(stackCut[1]) && stackCut[0].Symb == GrammarStart.Symb)
{
string postfixAdd = "";
if (vars.Count > 0)
{
postfixAdd = vars.Pop() + stackCut[1].Symb;

}
//else
//     postfixAdd = stackCut[1].Symb;

postfix += postfixAdd;

stack.Pop(); stack.Pop();
stack.Push(GrammarStart);

flag = false;
}
}
else if (stackCut.Count == 3)
{
```

```
if (stackCut[0].Symb == Parens[1].Symb && stackCut[1].Symb == Gramma
{
stack.Pop(); stack.Pop(); stack.Pop();
stack.Push(new Symbol(stackCut[1].Symb, stackCut[1].Value));

flag = false;
}
else if (stackCut[0].Symb == GrammarStart.Symb && IsOper(stackCut[1])
{
string postfixAdd = stackCut[1].Symb;
if (stackCut[0].Value != "")
postfixAdd = stackCut[0].Value + postfixAdd;
if (stackCut[2].Value != "")
postfixAdd = stackCut[2].Value + postfixAdd;

postfix += postfixAdd;

stack.Pop(); stack.Pop(); stack.Pop();
stack.Push(GrammarStart);

flag = false;
}
}
}

if (flag)
{
Console.WriteLine("Wrong construction at {0} pos!", currToken + 1);
return null;
}
}
else
{
Console.WriteLine("Error at {0} pos!", currToken + 1);
return null;
}
}

return postfix;
}

private Symbol NotStartSymbolHead(Stack<Symbol> stack)
{
for (int i = 0; i < stack.Count; i++)
{
if (stack.ElementAt(i).Symb != GrammarStart.Symb)
return stack.ElementAt(i);
}

return null;
```

```
}

private bool IsOper(Symbol symb)
{
return Ops.Any(o => o.Symb == symb.Symb);
}

private bool IsUnarOper(Symbol symb)
{
if (symb.Symb == "s+" || symb.Symb == "s-")
return true;

return false;
}

private bool IsNot(Symbol symb)
{
if (symb.Symb == "not")
return true;

return false;
}

private bool IsVar(string symb)
{
return Vars.Any(v => v.Symb == symb);
}

private void TableOutput()
{
Console.WriteLine("Precedence table:");

Console.Write("     | ");
for (int i = 0; i < symbols.Count; i++)
{
Console.Write("{0, 3} | ", symbols[i].Symb);
}
Console.WriteLine();

for (int i = 0; i < symbols.Count; i++)
{
Console.Write("{0, 3} | ", symbols[i].Symb);
for (int j = 0; j < symbols.Count; j++)
{
Console.Write("{0, 3} | ", precedenceTable[i][j]);
}
Console.WriteLine();
}
}
```

```
private bool IsStackDone(Stack<Symbol> stack)
{
if (stack.Count == 2)
if (stack.ElementAt(0).Symb == GrammarStart.Symb && stack.ElementAt(1
return true;

return false;
}

public static string[] ReplaceUnar(string[] input)
{
var tokens = input;

if (tokens[0] == "+")
{
tokens[0] = "s+";
}
if (tokens[0] == "-")
{
tokens[0] = "s-";
}

for (int i = 1; i < tokens.Length; i++)
{
if (tokens[i] == "+" && tokens[i - 1] == "(")
{
tokens[i] = "s+";
}

if (tokens[i] == "-" && tokens[i - 1] == "(")
{
tokens[i] = "s-";
}
}

return tokens;
}
}
}
```

3 Проверка правильности программы

Тестовые данные:

Входная строка	Результат
(a+c) * a <> a	ac+a*a<>
a-a+c	aa-c+
a-a div c	acdiva-
a-c mod a	camoda-

+ a	a+
-(a+c/a-a)	ca/a+a−
(a + c))	-
(a + c *)	-
a + c >=	-

4 Выводы

По результатам проведенной работы было проведено ознакомление с основными понятиями и определениями, лежащими в основе синтаксического анализа операторного предшествования. Был изучен алгоритм синтаксического анализа операторного предшествования, а также разработана, оттестирована и отлажена программа синтаксического анализа в соответствии с предложенным вариантом грамматики. В программу синтаксического анализа были включены семантические действия для реализации синтаксически управляемого перевода инфиксного выражения в обратную польскую нотацию.

5 Список литературы

1. АХО А.б УЛЬМАН Дж. Теория синтакстического анализа, перевода и компиляции: В 2-х томах. Т1.: Синтаксический анализ. - М.: Мир, 1978.
2. АХО А.В, ЛАМ М.С., СЕТИ Р., УЛЬМАН Дж.Д. Компиляторы: принципы, технологии и инструменты. – М.: Вильямс, 2008.